1 Einleitung

Fassen wir noch einmal zusammen, was wir über das Problem wissen, dann wird die Lösung sich von selbst aufdrängen.

Eduard von Hartmann (1842-1906): Philosophie des Unbewussten

1.1 Probleme bei der Software-Entwicklung

Die chronische Software-Krise – eine Krise, die sich ad infinitum fortzusetzen scheint

Es macht wenig Sinn, es den vielen Büchern und Vorträgen über Software-Engineering gleichzutun und im ersten Teil ein umfassendes Klagelied über die Probleme der IT-Branche anzustimmen, z.B. mangelhafte Termintreue, zahlreiche Fehler, geringe Akzeptanz der Anwender, Ineffizienz der Entwicklung etc.

Auch die Lösung für die Probleme drängt sich schon seit Jahrzehnten auf: Es ist hinlänglich bekannt, dass u.a. die fehlenden oder falschen Anforderungen und die »Herumbastelei« im Code ohne entsprechende Konzepte und Modelle »schuld« an der Misere sind. Die Software-Krise der 80er-Jahre ist mittlerweile chronisch [Gibb94], und die Diskussion um das optimale Vorgehen bei der Software-Entwicklung kommt nicht so recht voran. Zwar sind entsprechende Studien, z.B. die Chaos Reports [Stan04], vorhanden, unklar bleibt aber dennoch, wie es denn nun tatsächlich zu einer Verbesserung der unbefriedigenden Situation kommen kann.

Intensivierung der Modellierung, Reduktion der Programmierung auf Code-Ebene: die Evolution der Informatik?

Zwar gibt es nach Assembler, 3- und 4-GLs[1] sowie OO-Sprachen eine gewisse Evolution in Hinblick auf die Leistungsfähigkeit der Software-Entwicklung, aber die Komplexität der Applikationen (Verteilung, Sicherheit, Performanz) einschließlich der Plattformen erhöht sich ebenfalls. Vereinfacht ausgedrückt lässt sich die Modellierung als nächste (Sprach-)Stufe bezeichnen, um der wachsenden Komplexität

1. Vertreter der 3 GL (3rd Generation Langauge = Programmiersprache der 3. Generation) wären z.B. Pascal, Modula und C, während die 4 GL oft als Makro- oder Skriptsprachen bezeichnet werden, z.B. JavaScript.

der Systeme und Plattformen begegnen zu können. Dies ist die Motivation für modellgetriebene Ansätze wie MDA.

Obwohl eingangs das Versprechen abgegeben wurde, nicht wieder über all die Probleme der IT-Branche zu referieren, seien dennoch zwei Bereiche kurz erwähnt, da sich mit ihnen wichtige Festlegungen für dieses Buch verbinden.

Mangelhafte Strukturierung der Informationen

Die in der Analyse- und Design-Phase entstehende Menge an Dokumenten und das Fehlen von Festlegungen bzgl. der Struktur kann zu Inkonsistenzen führen – insbesondere bei der Arbeit im Team. Die Folge sind kaum auffindbare Detailinformationen und das Vergessen von Anforderungen bzw. deren fehlerhafte Umsetzung. Die Gültigkeit der Dokumente ist dann zu Recht von den Beteiligten in Frage zu stellen, und eine nachträgliche Pflege erfolgt dann wegen des Termindrucks oftmals nicht. Dem Projekt wird einerseits die fachlich fundierte Grundlage entzogen, andererseits aber auch jede juristische Absicherung dem Kunden gegenüber.

Artefakt = Ergebnis einer Tätigkeit im Rahmen der Software-Entwicklung

Im Folgenden wird statt des Begriffes *Dokument* der allgemeinere Begriff *Artefakt* verwendet, womit alle (Zwischen-)Ergebnisse der Software-Entwicklung – also auch der Code – gemeint sind und nicht nur Textdokumente[2].

Fehlende Architekturdefinitionen

Architekturen sollten im Rahmen des Software-Entwurfes (Design-Phase) eigentlich bewusst unter Beachtung von Qualitätsanforderungen erarbeitet werden. Leider entsteht die Software-Architektur oftmals zufällig oder aufgrund externer Gegebenheiten, die nicht immer förderlich sind.

Software-Architekturen sollten bewusst und konsequent im Sinne der Anforderungen an das System konzipiert werden.

Als seien die fehlenden Entwurfsentscheidungen für die Entstehung einer adäquaten Software-Architektur nicht schon schlimm genug, fehlen dann auch noch die Kenntnisse (oder Werkzeuge), um diese so zu definieren, dass es einen praktischen Nutzen hat: Auf dem Papier sind Software-Architekturbeschreibungen zwar schon »persistenter« als im Kopf, technisch be- und verwertbar sind sie jedoch nur schwer. So ist eine Code-Generierung auf Basis einer Skizze in einem Textdokument kaum möglich.

Bei der MDA liegen Software-Architekturen mitsamt den relevanten Plattformen formal definiert vor und lassen sich direkt im Entwick-

2. Der Artefaktbegriff findet sich u.a. beim IBM Rational Unified Process [IBMRUP] und ist mittlerweile sehr verbreitet.

lungsprozess verwenden, z.B. im Sinne einer automatisierten Code-Erzeugung.

Modellierung + Formalisierung = Lösung der Probleme?

Anforderungsdokumente und Architekturbeschreibungen, die in natürlicher Sprache bzw. in Form nichtformaler Modelle vorliegen, sind nicht maschinenlesbar bzw. interpretierbar und somit auch nicht (automatisch) transformierbar. Erst während des Designs bzw. der Implementierung kommen zwangsläufig formale Sprachen (z.B. Java) zum Einsatz (siehe Abb. 1–1).

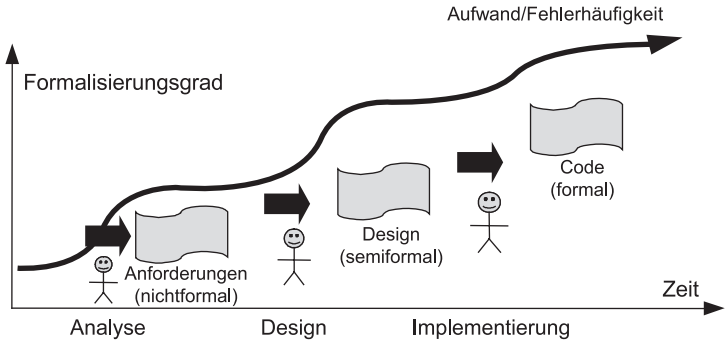


Abb. 1–1
Stetig steigende Aufwände durch geringe Formalisierung am Anfang.

Würde dagegen die Formalisierung früher, d.h. in der Analysephase, ansetzen, könnten nicht nur zahlreiche Fehler vermieden, sondern auch viele Arbeiten automatisiert werden, womit sich der Aufwand insgesamt vermindert – so weit die Theorie (siehe Abb. 1–2).

Hohe Formalisierung ermöglicht hohen Automatisierungsgrad.

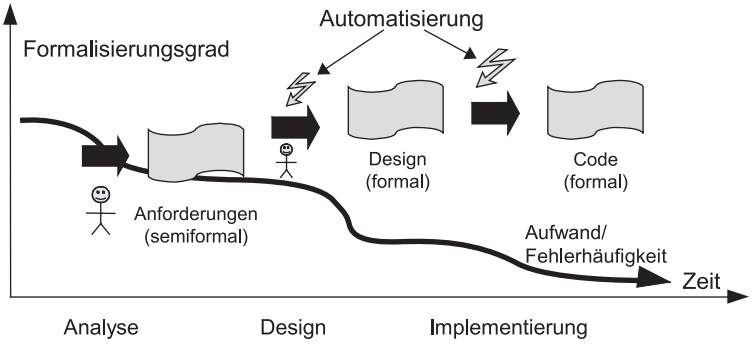


Abb. 1–2
Sinkende Aufwände durch hohe Formalisierung/ Automatisierung.

Eine zentrale Frage ist jedoch beim Thema Formalisierung offen geblieben: Was ist mit der Software-Architektur und der (Ziel-)Plattform? Auch sie müssen formal definiert vorliegen, damit eine automatische Weiterverarbeitung durch Transformationen stattfinden kann, die schließlich zum ausführbaren Code führt. Genau dies fordert MDA

Zentrale Begriffe von MDA: PIM, PSM, Architektur, Plattform

mit dem Konzept der plattformunabhängigen und plattformabhängigen Modelle. Im MDA-Sprachgebrauch nennt man sie *Platform Independent Model* (PIM) und *Platform Specific Model* (PSM): Die Software-Architektur wird genauso wie die Plattform(en) formal durch Modelle beschrieben. An dieser Stelle wird bereits deutlich, warum MDA weit über die einfache Code-Generierung hinausgeht.

Bevor nun der MDA-Ansatz erläutert wird, sei auf das Thema Artefakte und deren zentrale Bedeutung für die Qualität von Software-Systemen eingegangen.

Aufgaben

1. Nennen Sie zwei Beispiele, welche die chronische Software-Krise ausmachen.
2. Welchen Zusammenhang sehen Sie zwischen Formalisierung und Automatisierung?

1.2 Artefakte

Artefakte als Ergebnis einer Entwicklungstätigkeit können Modelle, Texte oder Code sein. Zusammengefasste Artefakte, die einen ganzen Themenkomplex, z.B. Anforderungen, abdecken, sind einerseits als Dokumente bekannt, z.B. als Anforderungsspezifikation (Requirements Specification), und andererseits als Software-Bausteine, z.B. Modul, Komponente, Subsystem.[3]

> Artefakte sind Arbeitsergebnisse (End- oder Zwischenergebnisse), die während eines Projektes produziert und benutzt werden. Artefakte werden verwendet, um projektspezifische Informationen festzuhalten oder zu übermitteln.

Die o.g. sehr allgemeine Definition ist allein wenig hilfreich, da so ziemlich alles, was mit der Software-Entwicklung im weitesten Sinne zu tun hat, als Artefakt zu bezeichnen ist. Interessant sind daher die verschiedenen Aspekte von Artefakten, z.B.:

- Zeitlicher Bezug: In welchem Projektabschnitt bzw. in welcher Phase oder welchem Arbeitspaket entsteht das Artefakt, und wann wird es wie verwendet?

3. Eigentlich bedeutet das aus dem Lateinischen stammende Wort *Artefakt* so viel wie *Kunsterzeugnis*. Der Begriff wurde von der Informatik übernommen und besonders durch die Verbreitung des RUP (Rational Unified Process, [IBMRUP], [Kruc04]) bekannt.

- Personeller Bezug: Wer ist verantwortlich für das Artefakt bzw. erstellt es oder hat Zugriff darauf?
- Medialer Bezug: In welcher Form wird das Artefakt erzeugt, und wie wird es gespeichert?

Die nahe liegende Vermutung, zwischen einem Artefakt und einem Dokument wäre eine 1:1-Beziehung, ist nicht zielführend und hätte besonders im Falle von Dokumenten in Papierform fatale Folgen: Statt verwertbare Ergebnisse zu produzieren, würde der Fokus auf dem Ausdruck liegen. Fazit: »Typically, artifacts are *not* documents« ([Kruc04], S. 41). Artefakte können jedoch in *Information Sets* gruppiert werden, z.B. enthält das *Requirements Set* alle anforderungsorientierten Artefakte (Use-Cases, nichtfunktionale Anforderungen etc.). Besonders mit dem in Abschnitt 2.2.3 eingeführten Konzept der *MDA Viewpoints* und *Views* (siehe S. 53) erscheint dies sinnvoll und hilfreich. Weitere *Sets* sind *Management Set, Design Set, Implementation Set* und *Deployment Set*.

Neben der o.g. eher phasenorientierten Sichtweise gibt es noch zwei wichtige Aspekt für Artefakte, die im Folgenden zu untersuchen sind und zu einer Klassifikation führen: Generierungs- und Transformationsaspekt.

Klassifizierung bezüglich der Generierung: manuelle und generierte Artefakte

Artefakte eines Software-Entwicklungsprozesses lassen sich durch den Aspekt der Aktualität bzw. der Art der Erzeugung in drei verschiedene Klassen unterteilen:

- **Manuell bearbeitete Artefakte** (kurz: **manuelle Artefakte**) sind alle (Zwischen-)Produkte, mit denen aktiv gearbeitet wird, z.B. dienen Analysemodelle als Kommunikationsmittel zwischen dem Anwender und der Software-Entwicklung. Sie sollten menschenlesbar, vollständig, konsistent sowie korrekt sein. Zu den manuellen Artefakten gehören auch Quellcode-Dateien, die manuell erstellt und bearbeitet werden.
- **Generierte Artefakte** sind zu 100% aus anderen generierten oder manuellen Artefakten automatisch erzeugt (generiert) und damit zu 100% redundant, z.B. Class-Files in einem Java-Projekt.
- **Historische Artefakte** sind lediglich dazu da, um die Vergangenheit zu dokumentieren, und haben in der Regel wenig praktische und hauptsächlich juristische Relevanz. Hierzu zählen Gesprächsprotokolle oder Konzeptskizzen, die bewusst nicht weiterentwickelt werden, also nur einen *Snapshot* einer bestimmten Entwicklungsstufe darstellen.

Artefaktklassifizierung für die MDA: Trennung von manuellen und generierten Artefakten

Fehlt die Trennung dieser drei Artefaktklassen, kommt es zu schwer beherrschbaren Redundanzen und Inkonsistenzen. Schon bei der Anforderungsdokumentation erfolgt in der Praxis oftmals schon der erste Schritt in den gesicherten Untergang: In der initialen Erstellungsphase eines Projektes werden die Anforderungsartefakte erstellt und sind natürlich zunächst noch konsistent. Ab einer gewissen Menge an unstrukturierten Informationen wird es dann aber nahezu unmöglich, die Informationen untereinander manuell konsistent zu halten.

Spätestens nach der Einführung der Software beim Kunden kommt es dann zur ersten Flut von Änderungswünschen, die prompt im Code bearbeitet werden: Dieses Verfahren heißt scherzhaft VHIT-Methode (»Vom Hirn ins Terminal«), denn ab jetzt entsprechen die meisten der Anforderungs- und Entwurfsdokumente nicht mehr der Realität (sprich: dem Code, der in Form eines Software-Produktes beim Kunden zum Einsatz kommt).

Grundsätzlich sollte daher gelten: Jedes manuelle Artefakt muss während seiner gesamten Lebensdauer inhaltlich vollständig konsistent zu allen anderen Artefakten sein, bis es durch Archivierung (oder Löschen) zum historischen Artefakt erklärt wird.

Artefakte und Redundanzen

Probleme durch Redundanzen in manuellen Artefakten: divergierende Inhalte und damit potenzielle Inkonsistenzen

Redundanz in Artefakten ist nicht grundsätzlich schlecht, aber zu vermeiden, wenn es in der Folge der Redundanz (potenziell) zu redundanten, manuellen Aktivitäten kommt, was i. d. R. die Minderung der Effizienz sowie Inkonsistenzen (und damit eine höhere Fehlerquote) bewirkt. Sekundäreffekte bleiben in diesem Fall nicht aus: Es entsteht Zusatzaufwand durch das Überwachen der Abhängigkeiten, denn wird ein Artefakt geändert, muss auch der redundante Teil im anderen Artefakt geändert werden.

Die Notwendigkeit, ein Software-Projekt aus verschiedenen *Perspektiven* (*Viewpoints*) auf unterschiedlichen *Abstraktionsebenen* zu beschreiben, führt unweigerlich zu Redundanzen. Problematisch sind Informationen einer höheren Abstraktionsebene, die implizit in den darunter liegenden Ebenen enthalten sind. Wird beispielsweise ein bestimmter Anwendungsfall (*Use-Case*) in einem Dokument zunächst grob beschrieben und daraufhin in einem weiteren Dokument genauer spezifiziert bzw. entworfen (*Use-Case Realization*), so entsteht eine erhebliche Redundanz dem ersten Dokument gegenüber (*vertikale Redundanz*). Hieraus ergeben sich zwei Kernprobleme im Falle einer manuellen Erstellung und Pflege der Artefakte:

- Je umfangreicher die Dokumentation (Anforderungen, Design etc.) ist, desto aufwändiger wird jede Änderung an der Software.
- Die Gefahr von Inkonsistenzen steigt ebenfalls überproportional. Damit kann ein großer Teil der Dokumente wertlos werden.

Lösung des Redundanzproblems

Vereinfacht gesehen, stehen zwei Mittel zur Verfügung, um den o.g. Problemen bei der Redundanz zu begegnen:

Lösung des Redundanzproblems: Nutzung von Frameworks und Automatisierung

- Auflösen der Redundanz durch Wiederverwendung und ggf. Modifikation »fremder« Lösungen: Diese Art der *Externalisierung*, d.h. das Verlagern bestimmter Bereiche einer Software, ist mittlerweile sehr gebräuchlich, z.B. mit Entwurfsmustern (*Design Patterns*) oder Rahmenwerken (*Frameworks*). Letztere decken bestimmte Aspekte wie z.B. Persistenz ab, so dass die abstrakte Beschreibung der zu spezifizierenden (eigenen) Artefakte völlig ausreichend ist, d.h., es genügt der Vermerk »persistent« bei einer Klasse – den Rest erledigt das Framework[4]. Redundante Artefakte lassen sich mit dieser Technik vermeiden. Der Nachteil ist allerdings die technische Abhängigkeit, z.B. von einem Framework, wobei evtl. die Flexibilität (Änderbarkeit, Portierbarkeit) der Software eingeschränkt wird.
- Automatisierung der Redundanz: Bei einer automatischen Verarbeitung muss Redundanz kein grundsätzliches Problem darstellen, da weder manueller Mehraufwand anfällt, das Werkzeug mit der Redundanz umgeht und auch keine Abhängigkeiten entstehen müssen. Ein Compiler beispielsweise arbeitet hochgradig redundant und erzeugt große Mengen redundanter Daten, was sich jedoch nicht negativ auswirkt, da der Code nicht manuell kompiliert und auch das Kompilat nicht manuell gewartet wird.

  Natürlich ist diese Lösung für das Redundanzproblem nicht »kostenlos« zu haben: Die Voraussetzung für die Automatisierung ist ein hoher Grad an Formalisierung. Auf diesen Begriff geht das nächste Kapitel näher ein.

4. Bei einer JDO-Implementierung (Java Data Objects) beispielsweise wird mit dem Persistence-Manager-Interface und der Methode `makePersistent()` für die Persistierung gearbeitet, während dies bei Hibernate mit dem Session-Interface und der `save()`-Methode erfolgt. Auch bei der Abfragesprache gibt es Unterschiede: einerseits kommt die JDOQL (JDO Query Language) zum Einsatz, andererseits sind es die HQL (Hibernate Query Language) bzw. native SQL-Abfragen. Um die Abhängigkeiten zu minimieren, müssen Gegenmaßnahmen ergriffen werden, z.B. Design Patterns (Fassade, Wrapper, Adapter etc.).

Eigentlich scheint damit das Redundanzproblem gelöst zu sein: Mit der Wiederverwendung und/oder Automatisierung lassen sich manuelle Artefakte »identifizieren«. Was aber passiert, wenn in einem automatisch erstellten Artefakt manuell Anteile eingefügt werden müssen? So kann es z.B. nach einer Code-Generierung notwendig werden, Geschäftslogik in eine Java-Methode einzufügen, oder in einem Framework, d.h., in externen Komponenten, müssen Veränderungen vorgenommen werden. Besonders im Falle von generativen Ansätzen sei daher noch zusätzlich zwischen Quell- und Zielartefakten unterschieden.

Klassifizierung bezüglich der Transformation: Ausgangs- und Zielartefakte

Unerwünschter Effekt: Zielartefakte mit manuellen und generierten Anteilen

Ausgangs- oder Quellartefakte dienen als Basis für die nachfolgende (Weiter-)Verarbeitung, bei der im Ergebnis Zielartefakte entstehen. Wird beispielsweise ein Fachklassenmodell für die Code-Erzeugung verwendet, entstehen Code-Module als Zielartefakte. Der Idealfall wäre dabei, dass das Ausgangsartefakt ein manuelles Artefakt ist und das Zielartefakt ein generiertes Artefakt darstellt, das »nur« redundante Informationen bezogen auf das manuelle Artefakt, d.h. das Fachklassenmodell, hat. Bei einer Ergänzung des generierten Codes entsteht jedoch ein »Zwitterartefakt« mit manuellen und generierten Anteilen.

Abbildung 1–3 zeigt ein solches Artefakt: Aus dem manuellen Quellartefakt (Klassenmodell mit der Klasse *Produkt*) entsteht per Code-Generierung eine Java-Datei als Zielartefakt. Da sich nur die Attribute als Instanzvariablen sowie die Methodensignatur (Methode `getMWSt()`) generieren lassen, ist der Methodenrumpf (Algorithmus) manuell zu programmieren – mitten in den generierten Code hinein.

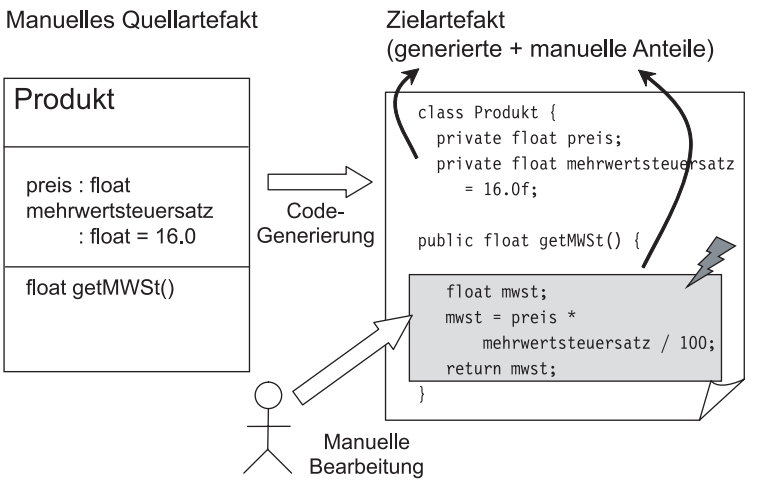


Abb. 1–3 *Problem der Vermischung bei der manuellen Bearbeitung von generierten Zielartefakten*

Eine strikte Trennung von manuellen und generierten Artefakten ist wünschenswert, ja sogar notwendig, da bei einer Vermischung verschiedene Folgeprobleme drohen. So könnten z.B. bei einer erneuten Generierung die manuellen Anteile des Zielartefaktes verloren gehen. Auch wäre bei einem Fehler unklar, welches Artefakt ursächlich war: das Ausgangsartefakt oder das später veränderte Zielartefakt.

Generiertes Zielartefakt: Es sollte keine manuelle Bearbeitung des Generates stattfinden.

Dem Ruf nach Isolation der manuellen von den generierten Teilen eines Zielartefaktes steht die Notwendigkeit eines Zusammenspiels entgegen: Eine generierte Methode braucht nun einmal einen Rumpf, der den Algorithmus enthält. Wird dieser nicht auf dem Ausgangsartefakt generiert, so kommt man um manuelle Teile nicht herum. Zwei Lösungen bieten sich an:

- Zielartefakte=100 % generierte Artefakte: Es finden sich keine manuell zu bearbeitenden Anteile in den Zielartefakten, was bedeutet, dass das Quellartefakt als manuelles Artefakt alle notwendigen Informationen enthält. Dies wäre sicherlich die effektivste Möglichkeit, sie lässt sich nur nicht in allen Fällen realisieren, da beispielsweise die Werkzeuge oder die Modellierungssprache noch unvollkommen sind.
- Die manuellen Anteile in Zielartefakten sind gekennzeichnet, isoliert und geschützt: Die Kennzeichnung sichert eine Identifikation, z.B. bei der Suche nach einer Fehlerursache (Debugging). Die Isolation ermöglicht eine separate Entwicklung bzw. Veränderung, und der Schutz sichert den Teil eines Artefaktes vor dem Verlust.

Probleme bei der Code-Generierung mit anschließender manueller Bearbeitung sind durch Trennung der Artefakte vermeidbar.

Abbildung 1–4 zeigt beispielhaft die zweite Problemlösungsstrategie bei der Code-Generierung auf der Basis eines Klassenmodells: Die generierten Zielartefakte sind abstrakte Klassen, die manuellen Zielartefakte müssen manuell in Form von Unterklassen erstellt werden. Einige Werkzeuge lösen das Problem mit *Protected Regions* (geschützte Bereiche im generierten Code), die im Abschnitt 3.3.3 näher erläutert werden (S. 135ff.)

Natürlich bleiben bei dieser Lösung offene Fragen, z.B. bei der Änderung der Klassennamen im Ausgangsmodell. Insofern ist dieser Kompromiss mit seinen Nachteilen nur dann zu wählen, wenn sich die 100 %-Lösung (Zielartefakte = generierte Artefakte ohne manuelle Anteile) nicht realisieren lässt.

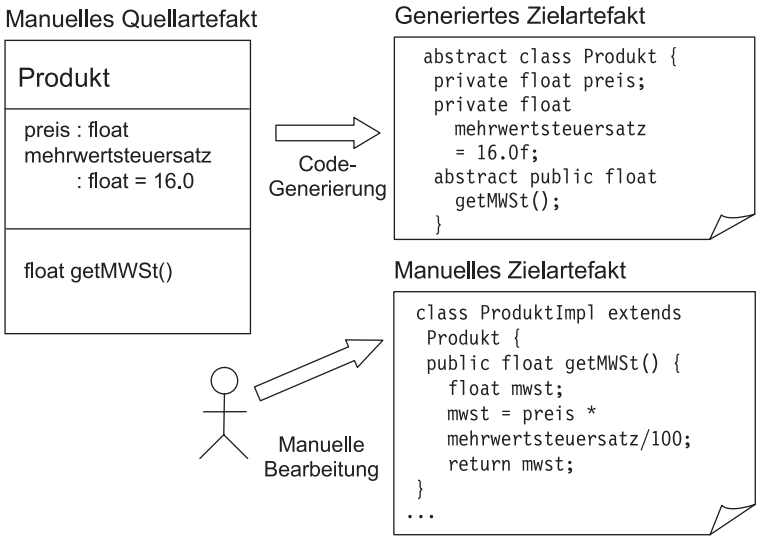


Abb. 1–4 *Problemlösungsstrategie bei manuellen Anteilen in Zielartefakte: Trennung der generierten von den manuellen Bereichen*

Fazit

Software-Entwicklung in zwei Dimensionen: Quell- und Zielartefakte sowie generierte und manuelle Artefakte

Zusammenfassend lässt sich feststellen, dass es im Falle von Redundanzen bei Artefakten – als eines der großen Probleme der Software-Entwicklung – einen interessanten Lösungsansatz mit der Automatisierung geben kann: Aus manuell bearbeiteten Ausgangsartefakten werden generierte Zielartefakte. Abbildung 1–5 zeigt neben dem problematischen Fall der manuellen Zielartefakte mit schwer beherrschbaren Redundanzen den Idealfall eines generierten Zielartefaktes. Aber auch die Mischform ist dargestellt, bei der es manuelle und generierte Anteile gibt. Ein Ausgangsartefakt kann ebenfalls manuelle und generierte Anteile haben, da es evtl. selbst als Zielartefakt aus einer Generierung bzw. manuellen Bearbeitung hervorgegangen ist.

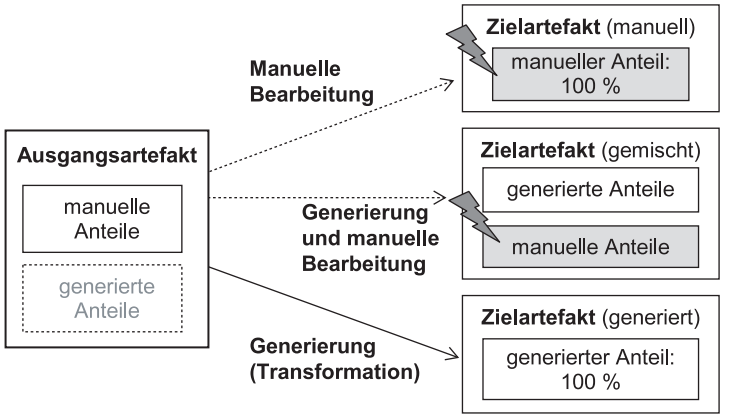


Abb. 1–5 *Ausgangs- und Zielartefakte mit manuellen und generierten Anteilen*

Die beiden Redundanzvermeidungsstrategien (Wiederverwendung und Automatisierung) bei Ausgangs- und Zielartefakten sind jedoch keineswegs als zwei sich ausschließende Alternativen zu sehen – im Gegenteil: Sie lassen sich sinnvoll im Rahmen der MDA kombinieren, wie später im Praxisteil zu zeigen ist.

Aufgaben

3. Es gibt drei Arten von Artefakten. Eine davon sind die so genannten *manuellen Artefakte*. Welche weiteren Arten kennen Sie?
4. Was ist das Problem bei der Redundanz? Machen Sie ein Beispiel für redundante Information in zwei UML-Diagrammen: Klassenmodell und State-Chart.
5. Das Auflösen der Redundanz durch Wiederverwendung (z.B. von Design Patterns) ist eine Möglichkeit, Redundanzprobleme zu lösen. Welche andere Methode kennen Sie? Vergleichen Sie beide Ansätze.

1.3 Formalisierung

1.3.1 Formale Sprachen

Natürliche Sprachen sind nicht eindeutig, formale Sprache schaffen Eindeutigkeit.

Formalisierung kann auf der Ebene der *Syntax* und der *Semantik* erfolgen. Daher seien zunächst diese beiden Begriffe näher betrachtet; anschließend wird dann auf die UML in Verbindung mit Formalisierung eingegangen.

Syntax einer Sprache

Zu unterscheiden sind *abstrakte* und *konkrete Syntax* einer Sprache: Die abstrakte Syntax (*Grammatik*) definiert die syntaktischen Elemente oder Konstrukte (z.B. Buchstaben) und regelt, wie Konstrukte (z.B. Wörter) aus anderen gebildet werden, während die konkrete Syntax die Ausdrucksmittel (Darstellungsform, Notation) festlegen, d.h., die konkrete Syntax gibt der abstrakten Syntax – und damit der Sprache selbst – ihr »Gesicht«. Die abstrakte Syntax der UML wird durch Metamodelle definiert, die konkrete Syntax durch natürlich-sprachliche Beschreibungen und Illustrationen (Näheres dazu in Abschnitt 2.3 ab S. 61).

Die Sprachsyntax existiert auf zwei Ebenen: abstrakt und konkret

Bei der Sprachdefinition kann es hilfreich sein, eine abstrakte Syntax zu schaffen, die unabhängig von der Darstellung ist, und eine konkrete Syntax, welche die Notation auf die abstrakte Syntax abbildet (Mapping). Dadurch ist die abstrakte Syntax wiederverwendbar und lässt sich leichter durch unterschiedliche Notationsformen darstellen.

Abstrakte und konkrete Syntax: Beispiel

Eine abstrakte Syntax mit Hilfe syntaktischer Regeln formal anzugeben, ist recht einfach. Abbildung 1–6 zeigt die abstrakte und konkrete Syntax in EBNF (Extended Backus-Naur-Form) für sehr einfache boolesche Ausdrücke: Die booleschen Werte `true` und `false` können durch die binären Operatoren `and` und `or` verknüpft werden. Die konkrete Syntax wird dabei auf die abstrakte Syntax abgebildet, was bei diesem Beispiel lediglich zu einer etwas besseren Selbsterklärbarkeit der Konstrukte führt, z.B. ist die konkrete Syntax für `bval` das Wort `boolean_value`.

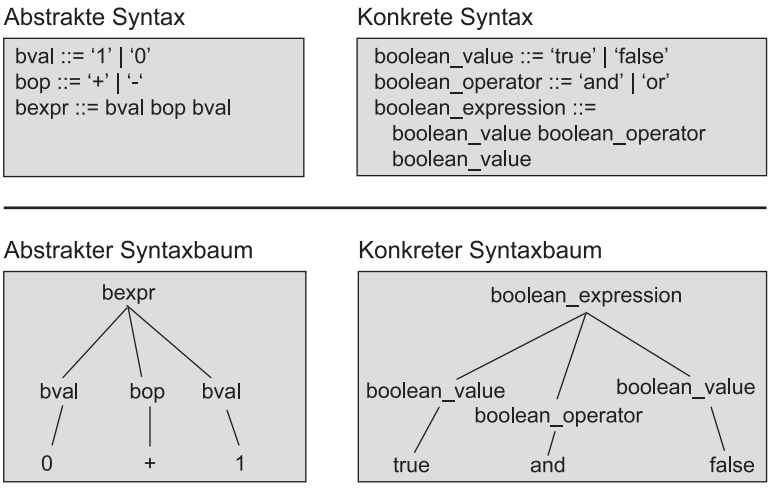


Abb. 1–6
Beispiel für eine abstrakte und konkrete Syntax mit Syntaxbaum

Die Syntaxbäume zeigen für den Ausdruck »`0+1`« bzw. »`true and false`«, auf welchen sprachlichen Konstrukten sie basieren, so dass prüfbar wird, ob ein solcher Ausdruck syntaktisch richtig ist (was in diesem Fall auch zutrifft). Ohne eine Syntax ließe sich dafür kein Syntaxbaum für einen konkreten Wert oder Ausdruck angeben.

Metamodellansatz: Grundlage der syntaktischen Korrektheit von Modellen

Das Thema *Modelle* und *Metamodelle* wird zwar erst im nächsten Kapitel erläutert, auf eine Analogie sei jedoch bereits hier hingewiesen: Mit einem *Metamodell* (abstrakte Syntax) lässt sich die syntaktische Korrektheit eines *Modells* (i.S. eines Syntaxbaums) prüfen.

Semantik einer Sprache

Weiterhin sind die statische (deklarative) und dynamische (operationale) Semantik festzulegen. Der statische Teil regelt, wie die Instanz eines Konstruktes mit einem anderen verbunden sein muss, um eine Bedeutung zu erhalten (Wohlgeformtheit). Die dynamische Semantik

legt die Bedeutung eines wohlgeformten Konstruktes fest. Nur wenn die Regeln der statischen Semantik erfüllt sind und damit eine Wohlgeformtheit vorliegt, definiert die dynamische Semantik eine Bedeutung für die sprachlichen Konstrukte.

Eine formale Sprache braucht eine definierte Syntax und Semantik, aber auch einen Konsens zwischen den Experten.

Allerdings muss als Voraussetzung für die Definition einer Semantik ein allgemein akzeptierter, so genannter *semantischer Geltungsbereich* existieren, dem ein »soziologischer Prozess« ([Hitz05], S. 322f.) vorausgeht, sprich: Es muss ein Konsens bei den (beteiligten) Menschen darüber vorhanden sein, um was es eigentlich geht. Eine Semantik für die boolesche Algebra beispielsweise ist nur dann definierbar, wenn Einigkeit darüber herrscht, was in der Mathematik eine Algebra ist und wieso dieser eine zweiwertige Logik zugrunde liegen kann.

Statische und dynamische Semantik: Beispiel

Da für das o.g. Beispiel noch keine Semantik festgelegt wurde, bleibt zunächst unklar, welche Bedeutung ein boolescher Ausdruck hat – insbesondere was das »Ausführungsergebnis« ist. Für den Ausdruck »true and false« gibt es daher zunächst keine Festlegung bezüglich des Wertes, zu dem er evaluieren soll. Dies ist nun mit der Semantikdefinition nachzuholen. Abbildung 1–7 stellt die statische und dynamische Semantik für das Beispiel vor.

Statische Semantik:

> Ein boolescher Ausdruck bestehend aus (boolean_value boolean_operator boolean_value) liefert einen booleschen Wert (boolean_value) zurück.

Dynamische Semantik:

> Ein boolescher Ausdruck bestehend aus (boolean_value boolean_operator boolean_value) evaluiert nur dann zu true, wenn beide booleschen Werte true sind, sonst zu false.

Abb. 1–7
Beispiel für die Definition der Semantik (statisch und dynamisch)

Statische und dynamische Semantik geben den Sprachkonstrukten eine Bedeutung.

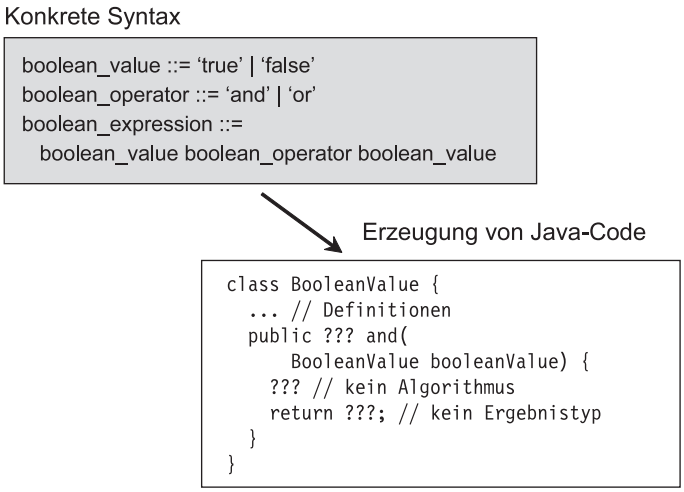
Die statische Semantik klärt die Wohlgeformtheit: Ein boolescher Ausdruck muss einen booleschen Wert liefern – sonst darf er sich nicht so nennen. Diese Festlegung kann erfolgen, ohne dass dabei eine Ausführung des sprachlichen Ausdrucks notwendig wäre. Die dynamische Semantik hingegen bildet die syntaktischen Elemente auf den semantischen Geltungsbereich ab, indem die erlaubten Ausführungen dargelegt werden. Hier geht es darum, zu was ein boolescher Ausdruck evaluiert. Abbildung 1–7 formuliert die Semantik auf einer natürlichsprachlichen Ebene und kann daher zwar keine formale Semantik lie-

fern, für die manuelle Anwendung der Verarbeitung eines Ausdrucks ist sie jedoch geeignet.

Formale Sprache: Syntax sowie statische und dynamische Semantik sollten im Idealfall vollständig formal vorliegen.

Es wäre also nicht möglich, ablauffähigen Programmcode für die nur auf der syntaktischen Ebene formal spezifizierte Sprache automatisch zu generieren, da dem Generator die Semantik nicht »klar« ist, denn woher soll er »wissen«, dass ein Ausdruck einen booleschen Wert ergibt: Die Syntax legt dies nicht fest. Folgendes Bild verdeutlicht diesen Sachverhalt am Beispiel von Java-Code. Es wird versucht, auf der Basis der Syntax eine Java-Klasse `BooleanValue` zu erzeugen, was nur im Ansatz erfolgreich ist. Noch nicht einmal eine komplette Signatur der Methode `and()` lässt sich formulieren, da – wie bereits festgestellt – die formale Semantik fehlt.

Abb. 1–8
Beispiel für die Generierung des Java-Codes auf der Basis der Syntax



Formalisierung und die UML

Die UML verwendet für die Definition der (abstrakten) Syntax einen Metamodellansatz: Mit UML-Klassenmodellen (also einem Teil der UML selbst) wird ein UML-Metamodell definiert, um die abstrakte Syntax der UML formal festzulegen. Dieser metazirkuläre Ansatz[5] verwirrt auf den ersten Blick, wird aber später noch eingehender erläutert. Wichtig ist an dieser Stelle: UML-Modelle sind Instanzen des UML-Metamodells. Durch diese Instanziierung kann die syntaktische Kor-

5. Die UML-Infrastruktur erklärt *metacircular* so*:* »In order to understand the description of the UML semantics, you must understand some UML semantics.« ([UML2Infra], S. 33) Einen offensichtlichen Vorteil hat das Ganze: Es ist kein anderer Standard notwendig, um die UML zu definieren; er ist eine »*self-contained specification*«.

rektheit (automatisch) geprüft werden. Die konkrete Syntax ist natürlich-sprachlich und durch grafische Darstellungen mit Beispielen festgelegt, d.h., lediglich die abstrakte Syntax der UML ist formal definiert.

Die formale Ebene für die Definition der UML besteht aus Metamodell und Metametamodell.

Die statische Semantik lässt sich mit der Object Constraint Language (OCL) ausdrücken [UML2OCL]. Besonders wenn die grafischen Möglichkeiten eines UML-Modells nicht mehr ausreichen, kommt sie zum Einsatz: Das folgende Klassenmodell spezifiziert, dass zu einem Termin ein Startdatum und ein (zeitlich dahinter liegendes) Enddatum gehören. Abbildung 1–9 zeigt zwei Darstellungsalternativen: Auf der linken Seite ist der Constraint (Zusicherung) unterhalb des Klassennamens innerhalb des Klassensymbols (Rechteck) dargestellt, auf der rechten Seite befindet sich der Constraint in einer Notiz, die mit der Klasse verbunden ist.

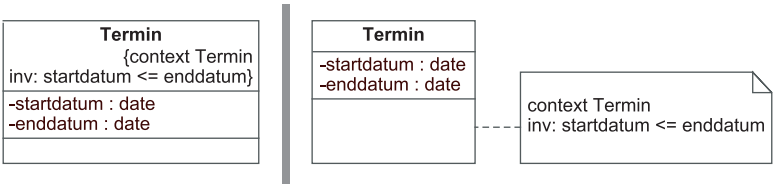


Abb. 1–9
OCL als Ergänzung bei der Definition der statischen Semantik (Beispiel)

Allerdings liegt die Semantik der UML nicht vollständig formal vor. Im Gegenteil: neben der OCL finden sich umfangreiche natürlich-sprachliche Beschreibungen, über deren Präzisionsgrad man sich streiten kann[6].

Die dynamische Semantik ist bei der UML nicht durchgängig formal.

Nichtsdestotrotz ist die OCL als formale Sprache geeignet, eine Schlüsselfunktion bei der MDA einzunehmen: Sie hilft bei der Präzision der Modellbeschreibung, lässt sich für Transformationen verwenden und bildet eine Basis bei der Definition einer (neuen) Sprache (vgl. [Warm03], S. 93ff.), so dass ein Einsatz sowohl auf Modell- als auch auf Metamodellebene sinnvoll ist. Abschnitt 2.3.1 geht etwas näher auf die OCL ein (siehe S. 61ff.).

Reifegrade von Modellen

OCL als wichtiger Baustein bei der MDA, z.B. für die Präzisierung von Modellen

In diesem Zusammenhang wird in Anlehnung an die Reifegrade des CMMI[7] von so genannten Model Maturity Levels (MML) gesprochen: Level 0 würde dabei bedeuten, dass eine Software auf keinerlei Spezifi-

6. »Oftmals würde man sich in diesen verbalen Beschreibungen [...] mehr Details und noch mehr Beispiele wünschen« ([Hitz05], 323). Der Autor erwähnt jedoch auch, dass zumindest im Falle von *State-Charts*, die ja Teil der UML sind, eine operational definierte Semantik vorliegt.
7. CMMI steht für *Capability Maturity Model Integration* und ist ein Reifegradmodell u.a. für Software-Entwicklungsprozesse vom SEI (Software-Engineering Institute) der Carnegie Mellon University in Pittsburgh, USA.

kation basiert, während Level 5 die vollständige, konsistente, detaillierte und präzise Beschreibung bedeutet ([Warm03], S. 9ff.). In der Praxis finden sich häufig natürlich-sprachliche Beschreibungen und Diagramme, die sich als (semi-)formale Modelle bezeichnen lassen (Level 2 und 3). Die MDA zielt klar auf den Level 4 mit präzisen Modellen, die eine direkte und automatisierte Transformation z.B. von Modell zu Code zulassen. Das Streben nach und das Erreichen von Level 5 hätte zwar zahlreiche Vorteile, fraglich ist jedoch, ob die beteiligten Menschen mit diesem Formalisierungsgrad umgehen können. Hier bedarf es noch eingehender Untersuchungen und Entwicklungen – insbesondere im Tool-Bereich.

Ausdrucksvolle Sprachen

Reifegrade von Modellen korrespondieren mit dem Formalisierungsgrad.

Bei formalen Sprachen ist schließlich noch zwischen semantisch ausdrucksvollen und weniger ausdrucksvollen Sprachen zu unterschieden: Die IDL (Interface Definition Language) von CORBA[8] beispielsweise ist eine semantisch schwache Sprache, da die Schnittstellenbeschreibungen kaum semantische Informationen enthalten, z.B. in Form von Pre- und Post-Conditions.

Bei der MDA ist es wichtig, keine »semantically thin«, sondern eine »semantically rich language« ([Fran03], S. 33f.) wie die UML zu verwenden, denn je höher die Ausdruckskraft der Quell- bzw. Ausgangsartefakte, desto größer die Möglichkeit, daraus nahezu vollständig die Zielartefakte zu erzeugen. Die Definition der Semantik ist daher eng mit der Ausdruckskraft einer Sprache gekoppelt.

Modellierungs- und Programmiersprachen haben einen unterschiedlichen Anwendungsbereich und unterscheiden sich in ihrer Ausdrucksmächtigkeit.

Damit ist ansatzweise auch die Frage beantwortet, warum besonders grafische Modellierungssprachen wie die UML im Vorteil gegenüber Programmiersprachen sind: Bei ihnen ist der Formalisierungsgrad flexibel »einstellbar«, und sie besitzen eine semantische Ausdruckskraft, die Programmiersprachen in aller Regel nicht haben (müssen).

Das Klassenmodell in Abbildung 1–10 zeigt, wie flexibel die Modellierung der Klasse `Gruppe` erfolgen kann: Auf der linken Seite ist eine grobe Darstellung ohne Datentypen und Operationen, während die rechte Seite einen hohen Grad an Modellierungsdetails aufweist. So ist z.B. für das Attribut `name` der Klasse `Gruppe` eine Zusicherung (Constrai[9]nt) in OCL angegeben, die fordert, dass dieser Name nicht leer

8. Der CORBA-Standard der OMG soll heterogene Software-Systeme interoperabel machen und kann als Vorläufer von Web-Services und SOA (Service Oriented Architecture) verstanden werden. CORBA steht für Common Object Request Broker Architecture.
9. Der Begriff *Constraint* kann auch als *Einschränkung* übersetzt werden. Im Rahmen des Buches soll es jedoch einheitlich beim Begriff *Zusicherung* bleiben.

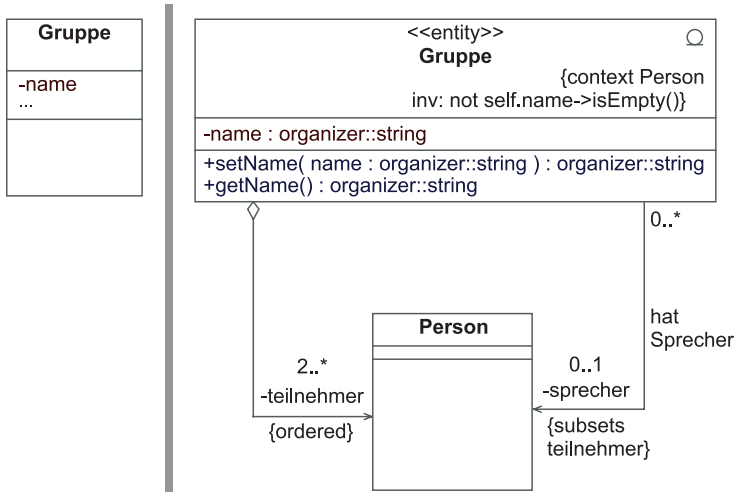


Abb. 1–10
Flexibilität und semantische Ausdruckskraft von UML

sein darf: `not self.name->isEmpty()`. Die Ausdrucksmächtigkeit wird auch durch die Aggregation und die Assoziation deutlich: Die Gruppe besteht aus mindestens zwei Teilnehmern, deren Reihenfolge bekannt ist (`ordered`-Constraint), und optional aus einem Sprecher, der aus dem Kreis der Teilnehmer stammt (Constraint `{subsets teilnehmer}`).

Modellierungssprachen sind für die Semantik, Programmiersprachen für die Ausführung geeignet.

Würde man dies in Java umsetzen, ginge ein Teil der Ausdruckskraft und damit der Semantik des Modells verloren. So gibt es z.B. bei der Deklaration der beiden Instanzvariablen `teilnehmer` und `sprecher` (zunächst) keine Möglichkeit, das Subset-Constraint programmtechnisch umzusetzen. Solche deklarativen Ergänzungen waren in der Vergangenheit nicht oder nur rudimentär vorhanden, so dass sich die Prüfung auf Einhaltung des Constraints »verstreut« in den Methoden der Klasse wiederfindet – auf Kosten der Lesbarkeit, der Wartbarkeit und vor allen Dingen der Testbarkeit, denn eine Prüfung gegen die Anforderungen im UML-Diagramm kann nur aufwändig per Hand erfolgen (siehe folgendes Listing). [10]

10. Es gibt bereits verschiedene Ansätze, dieses Problem zu lösen, z.B. auf der Basis von *Annotations*, die es seit Java 5.0 gibt. Mit entsprechend ausgereiften Werkzeugen könnte es hier in Zukunft eine breite Anwendung geben. Ein interessanter und vielversprechender Ansatz ist das *Extended Static Checking* (ESC) mit Hilfe der *Java Modeling Language* (JML). Das Verifikationswerkzeug ESC/Java verarbeitet Java-Programme, die mit der JML annotiert wurden. Das Werkzeug führt dann auf der Basis von Bedingungen und einem Theorembeweiser die Verifikation durch (vgl. [Flan02], [Cok04], [Kini05]).

Umsetzung der Klasse Gruppe als Java-Programm

```
class Gruppe {
   private List<Person> teilnehmer =
      new ArrayList<Person>();
   private Person sprecher;
...
   public void setSprecher(Person sprecher) {
      if (teilehmer.contains(sprecher)) {
         this.sprecher = sprecher;
...
```

Eines wird durch das o.g. Listing ebenfalls deutlich: Java bietet nicht wie die UML verschiedene Formalisierungsstufen an, was zwar theoretisch denkbar, praktisch aber zumindest ungewohnt, wenn nicht sogar unpraktikabel wäre. In einem UML-Klassenmodell müssen die Attribute der Klassen nicht zwingend typisiert sein, da es in frühen Phasen der Software-Entwicklung eher darauf ankommt, die Termini zu finden und als Domänenklasse zu modellieren, als diese sofort detailliert mit ihren Datentypen zu definieren. Bei einem Java-Programm hingegen leistet der Compiler sofort heftigen Widerstand und verweigert die Übersetzung, wenn eine Variable ohne Typ deklariert wird.

Verschiedene Modellierungs- und Programmiersprachen können im Rahmen der MDA zum Einsatz kommen.

Diese o.g. Feststellungen sollen nicht als Vorwurf oder negative Kritik gegenüber Java oder anderen Programmiersprachen missverstanden werden, denn diese Sprachen hatten nie einen anderen Anspruch (es sind halt Programmiersprachen). Ziel der Ausführungen war es, die beiden Sprachkonzepte, UML einerseits und Programmierung andererseits, gegeneinander abzugrenzen und die »Arbeitsteilung« deutlich zu machen, um ein zentrales Konzept der MDA nachvollziehen zu können: Statt Semantik einer Anwendungsdomäne mit einer semantisch schwachen Programmiersprache umzusetzen, sollte eine der Domäne angemessene, semantisch ausdrucksstarke Sprache zum Einsatz kommen – dies kann, muss aber nicht die UML sein, wenn sich eine andere domänenspezifische Sprache anbietet.

1.3.2 Formalisierung bei Artefakten der Software-Entwicklung

Bei der Formulierung von komplexen Zusammenhängen in natürlicher Sprache (Prosa) ist die unscharfe Definition der Semantik der einzelnen Sprachelemente zunächst kaum ein Problem, da der Mensch daran gewöhnt ist, mit dieser »Ungenauigkeit« umzugehen. So ist der Satz »Ein Katalog besteht aus verschiedenen Produkten« durchaus verständlich, auch wenn er – formal gesehen – semantisch falsch ist, denn ein Katalog besteht natürlich nicht aus den (realen) Produkten, sondern aus einer Anzahl von beschreibenden Elementen, z.B. Produktname als Text oder Bild.

Der Begriff *Produkt* hat demnach zwei Bedeutungen: Einerseits steht er als Abstraktion tatsächlich vorhandener Produkte und andererseits steht er für den Terminus, der eine Menge von Beschreibungselementen (z.B. Name, Preis, Größe) repräsentiert.

Natürliche Sprache: Die Interpretation von Prosa mit ungenauer Semantik beherrschen Menschen gut, Maschinen aber eher schlecht.

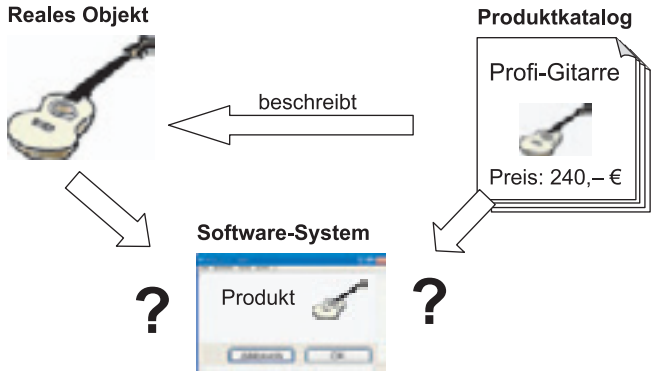


Abb. 1–11
Semantische Lücke am Beispiel des Produktkataloges

Die beiden (umgangssprachlich korrekten) Aussagen »Das Produkt ist im Katalog« und »Ich trage das Produkt nach Hause« verdeutlichen die Unschärfe, mit der die Software-Entwicklung umgehen muss: Eine Webanwendung für die Verwaltung von Produktkatalogen wird kaum mit den realen Produkten, sondern nur mit dem Terminus »arbeiten«.

Probleme: Semantische Lücken und Pragmatik

Da der Programmcode einen hohen Formalisierungsgrad hat, die Aussagen, die zur Entwicklung eben dieser Software führten, jedoch nicht, kommt es zu einer *semantischen Lücke*: Es ist oftmals unklar, wie die Transformation der Termini in die formale Welt der Software erfolgen soll[11]. So stellt sich bei dem Produktbeispiel (siehe Abb. 1–11) die Frage, was genau ein *Produkt* im Sinne der Software ist: Welche Produkteigenschaften sind bei der Software zu übernehmen bzw. abzubilden? Wie ist die Beziehung zwischen dem *Produkt* in der Software und dem realen Objekt herzustellen? Bleiben solche Fragen ungeklärt, steigt damit die Wahrscheinlichkeit, dass es Missverständnisse und Fehler bei der Umsetzung gibt.

11. Hier wird die Schwäche von in natürlicher Sprache formulierten Spezifikationen klar: Die einzelnen Sprachelemente sind in ihrer Semantik häufig mehrdeutig, und oft ist der Kontext einer Aussage nicht klar definiert, so dass ein Interpretationsspielraum entsteht. Dies wird bei dem Versuch deutlich, natürliche Sprache durch eine Maschine interpretieren zu lassen.

Semantische Lücken bei der Entstehung von Software erhöhen das Risiko von Fehlinterpretationen.

Zur Problematik auf der semantischen Ebene können sich Schwierigkeiten mit der *Pragmatik* hinzugesellen, die sich oft schon frühzeitig bemerkbar machen: Bei dem Katalogbeispiel stellt sich die Frage, was denn eigentlich eine Produktbeschreibung im Katalog ausmacht. Die Antwort »Ein Produkt hat eine Bezeichnung und einen Preis« ist zwar auf den ersten Blick hilfreich und lässt sich einfach auf einer Webseite unterbringen, aber kaum ist die Webanwendung dann fertig programmiert, fällt dem Kunden sofort ein, dass diverse Eigenschaften, z.B. die Produktabmessungen, fehlen, denn diese sind ja beim realen Produkt ebenfalls existent. Natürlich hätte dies schon vorab bemerkt und festgelegt werden können, es ist aber bei komplexen Systemen eher unwahrscheinlich, dass nach einer ersten Analyse wirklich an alles gedacht wurde und später keine Änderungen bzw. Ergänzungen auftreten. Ebenfalls völlig unpraktikabel wäre es, einfach alle Eigenschaften des Produktes in die Software zu übernehmen. Die Mehrzahl würde kaum benötigt, denn die Pragmatik bei der Verwendung eines Produktes in der Software ist eine andere als bei der Verwendung in der Realität: Eine Gitarre kann (gut) gespielt werden, während eine Software diese »nur« logisch verwaltet oder darstellt.

Syntax, Semantik und Pragmatik bei modellierten Gegenstandsbereichen

Die Praxis zeigt, dass der Übergang von der Realität zur (virtuellen) Welt der Software-Systeme i.d.R. kein linear sequenzieller Weg ist, der in einer Iteration abgeschlossen ist, sondern permanente Änderungen und Ergänzungen eher die Normalität als die Ausnahme darstellen. Findet diese Fluktuation an mehreren Stellen (z.B. Software-Entwurf, Code) gleichzeitig und manuell statt, kommt es zu den hinlänglich bekannten Problemen, welche die chronische Software-Krise ausmachen.

MDA: Syntax und Semantik einer Modellierungssprache müssen definiert werden.

Aus diesem Grund versucht die MDA, durch den Einsatz einer formalen Sprache den Interpretationsspielraum bei der Definition von Software-Systemen von Anfang an zu minimieren. Eine formal eindeutige und vollständige Definition von Syntax und Semantik für eine Sprache kann zumindest teilweise auf der Metaebene stattfinden. Wie bereits im vorherigen Kapitel erwähnt, gilt dies auch für die UML.

Formalisierung führt daher zu zwei sehr wichtigen Effekten: Einerseits eliminiert sie Interpretationsspielräume (Eindeutigkeit), womit semantische Lücken minimiert werden, andererseits hilft sie beim Weglassen nicht relevanter Informationen durch Isolation bzw. Abstraktion. Um diese eher theoretischen Betrachtungen anschaulicher zu machen, schauen wir uns im Folgenden zwei Beispiele für die Festlegung der Syntax und der Semantik von Artefakten an.

Formalisierungsbeispiel #1

Folgendes Beispiel zeigt das Ergebnis einer Formalisierung anhand der Plausibilität eines Datumsbereiches. Auf der linken Seite in Abbildung 1–12 befindet sich die natürlich-sprachliche (nicht formalisierte) Beschreibung, auf der rechten Seite eine formale Definition mit Hilfe der OCL.

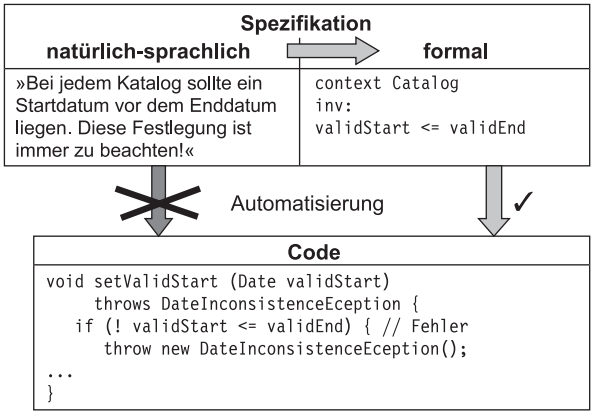


Abb. 1–12
Generierbarkeit von Code durch formale Spezifikation am Beispiel

Abgesehen davon, dass eine Code-Generierung bei rein natürlich-sprachlichen Beschreibungen kaum möglich ist, bleibt unklar, ob das Enddatum identisch mit dem Startdatum sein darf (semantische Lücke). Dies ist bei der formalen Variante klar definiert durch den »≤«-Operator, so dass keine Fehler bei der Interpretation entstehen.

Selbstverständlich ist die natürlich-sprachliche Formulierung von Anforderungen (noch) unabdingbar und kann i.d.R. nicht vollständig durch formale Konstrukte ersetzt, sondern nur auf diese abgebildet werden, d.h., formale Spezifikationen übernehmen die Aufgabe, einen möglichst großen Teil der ungenauen, natürlich-sprachlichen Aussagen auf eine eindeutige Ebene zu bringen, um eine automatische Prüfung und Weiterverarbeitung zu ermöglichen.

Prozess der Formalisierung

Ein Teil der Formalisierungsarbeit ist das Finden einer gemeinsamen, formalen und ballastfreien Sprache, um Sachverhalte gleicher Struktur syntaktisch gleichermaßen auszudrücken. Diese für den betrachteten Gegenstandsbereich und den damit agierenden Menschen passende Sprache nennt man *domänenspezifische Sprache* (DSL – *Domain Specific Language*). Für die Software-Entwicklung kommen textuelle bzw. grafische Modellierungssprachen zum Einsatz, z.B. die UML. Wie

bereits erläutert, ist unsere natürliche Sprache für diese Zwecke jedoch nur bedingt geeignet, sie muss allerdings eine Brückenfunktion wahrnehmen: Zusätzliche Erklärungen als Verständnishilfe haben zwar technisch gesehen »nur« ergänzenden Charakter, können jedoch wertvoll für die am Projekt Beteiligten sein, die keine oder wenig Routine beim Umgang mit formalen Spezifikationen haben.

Formalisierung kann nicht nur für Produkte, sondern auch für Prozesse angewendet werden.

Formalisierung bedeutet also die Definition und Beachtung formaler Regeln auf der Basis einer präzisen (formalen) Sprache, wobei sich nicht nur die fachlichen und technischen Anforderungen, sondern andere Aspekte der Software-Entwicklung formalisieren lassen, z.B. der Software-Entwicklungsprozess in Form von Vorgehensmodellen.

Ein erster Schritt in Richtung Prozessformalisierung: Vorgehensmodelle für die Software-Entwicklung

An dieser Stelle soll keine Grundsatzdiskussion über Vorgehensmodelle geführt werden, aber es sei darauf hingewiesen, dass die prozessorientierte Formalisierung oftmals an eine Grenze stößt, die im Falle der Artefakte, also bei der produktorientierten Formalisierung, nicht (mehr) in dieser Form vorhanden ist: Software-Entwicklungsprozesse sind allzu oft natürlich-sprachlich beschrieben bzw. durch semiformale Modelle (z.B. UML-Activity-Diagramme) ungenügend und praxisfern abgebildet. Vorgehensmodelle ohne (hilfreiche) Verbindung zur Software-Methodik haben daher zu Recht ein Akzeptanzproblem. In Lehrbüchern finden sich zwar detaillierte Beschreibungen, z.B. »Man beginnt mit der Identifizierung von Objekten und Klassen und bestimmt dann Attribute und Assoziationen« ([Wint05], S. 201), aber es bleibt ein erheblicher Anteil an zusätzlichen Kenntnissen und Erfahrung, um korrekte Modellierungsarbeit zu leisten, was sich auch durch Heuristiken zeigt (vgl. [Wint05], 281ff., [Shne97], [Riel96]).

Fazit: Vorgehensmodelle oder Vorgehensweise machen nur dann einen Sinn, wenn diese in Methoden des Software-Engineerings »zum Leben erweckt« werden, sonst erfüllen sie höchstens eine Alibifunktion für ein wenig ergiebiges Prozess-Assessment. Aber nun zurück zur produktorientierten Betrachtung in Verbindung mit Formalisierung.

Formalisierungsbeispiel #2

Hier sei nun kurz gezeigt, wie (schnell) sich bei der objektorientierten Modellierung und der damit verbundenen Formalisierung die Metaebene bemerkbar macht. Bei der Anforderungsanalyse muss der Systemanalytiker erkennen, welche Termini sich als relevante Domänenklassen abbilden lassen, welche Attribute diese Klassen haben und welche unberücksichtigt bleiben. Es sei angenommen, dass ein (zukünftiges) Software-System Kunden und Artikel verwalten soll. Dazu existiert folgende Aussage:

»Kunden haben die Attribute Kundennummer, Name und Anschrift. Ein Artikel besitzt die Eigenschaften Artikelnummer, Bezeichnung und Preis. Gute Kunden erhalten spezielle Preise für bestimmte Artikel.«

Formalisierung und automatische Transformation: ein weiteres Beispiel

Als Domänenklassen bieten sich *Kunde* und *Artikel* an. Auch die meisten Attribute sind schnell zugeordnet. Würde der Analytiker dies als XML-Datei dokumentieren, so könnte das wie folgt aussehen (Domänenklassen sind mit »entity« gekennzeichnet):

XML-Dokument für die Entitäten Kunde und Artikel

```
<entity name="Kunde">
   <property>Kundennummer</property>
   <property>Name</property>
   <property>Anschrift</property>
</entity>
<entity name="Artikel">
   <property>Artikelnummer</property>
   <property>Bezeichnung</property>
   <property>Preis</property>
</entity>
```

Fach- oder Domänenklasse formalisieren die Anforderungen aus der realen Welt der Anwender.

Jede Domänenklasse hat einen Bezeichner, d.h. einen Namen (z.B. »Kunde«, »Artikel«), und eine Liste von Eigenschaften. Bereits an dieser Stelle lässt sich die Metaebene erkennen: Eine Domänenklasse mit Attributen wird durch Metaobjekte repräsentiert (siehe Abb. 1–13).

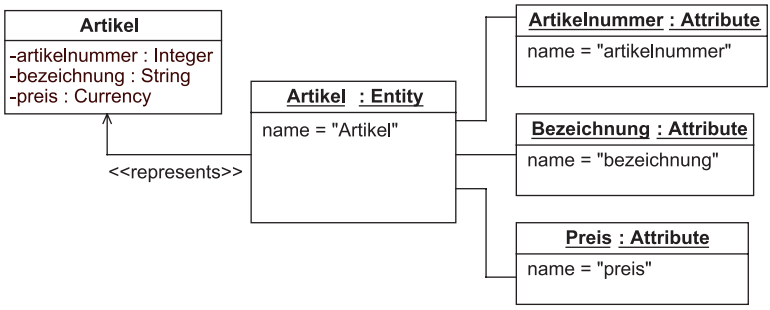


Abb. 1–13
Die Klasse Artikel repräsentiert durch Metaobjekte

Letztendlich drücken das XML-Dokument und das UML-Modell denselben Sachverhalt aus. Es stellen sich jedoch zwei Fragen: Wo kann der kundenspezifische Preis eines Artikels als Attribut modelliert werden? Wie sieht das zugehörige Metaklassenmodell aus? Die erste Frage ist im Rahmen der Aufgaben am Ende des Kapitels zu lösen, und die zweite Frage muss leider noch bis zum nächsten Kapitel offen bleiben. Wichtig war es aber an dieser Stelle, die für die MDA so wichtige Metaebene kennen zu lernen.

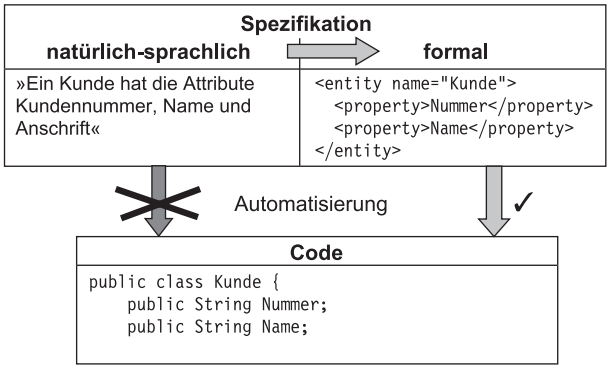
Was muss formalisiert werden?

Für die modellgetriebene Software-Entwicklung sind drei Bereiche von der Formalisierung betroffen: die Anforderungen, die Plattform des Zielsystems sowie die Transformationen. Die Anforderungen sind hierbei als Quelle (Ausgangsartefakt) und das Zielsystem als Produkt des Software-Entwicklungsprozesses (Zielartefakt) zu sehen, d.h., die Anforderungen werden in ein Software-System transformiert.

Automatisierung durch formale Modelle für die Anforderungen, die Plattform und die Transformationen

Mit dem Formalisierungsgrad lässt sich auch der Automatisierungsgrad steigern, im Sinne einer werkzeuggestützten Transformation. In Abbildung 1–14 ist eine natürlich-sprachliche Spezifikation einer formalen Spezifikation gegenübergestellt, wobei eine Code-Generierung nur bei Einsatz einer formalen Sprache möglich ist.

Abb. 1–14
Generierbarkeit von Code am Beispiel der Kundenklasse



Eines sollte durch das o.g. Beispiel jedoch auch deutlich geworden sein: Formalisierung heißt nicht Ersetzen von natürlich-sprachlichen Artefakten, sondern Ergänzung: Statt aus einem Text mit Anforderungen direkt Code zu erzeugen, werden ergänzende formale Konstrukte, die aus dem Text abgeleitet sind, dazu verwendet, die Code-Erzeugung zu automatisieren und durch Eindeutigkeit der formalen Anteile qualitativ zu verbessern. Natürlich-sprachliche Aussagen sind also nicht überflüssig, sondern werden wegen der besseren Verständlichkeit besonders für Nichtinformatiker benötigt.

Aufgaben

6. Was bedeutet die Formalisierung der Syntax bzw. der Semantik? Machen Sie ein Beispiel.
7. Wie kann beim Beispiel 2 der kundenspezifische Preis eines Artikels als Metaobjekt modelliert werden? Hinweis: Sie brauchen zusätzlich noch die Assoziationen als Metaobjekte.

1.4 Zusammenfassung

Die Software-Krise ist noch nicht überwunden: Wir befinden uns mitten drin.

Die seit über 30 Jahren bekannten Probleme der Software-Entwicklung sind noch nicht vollständig überwunden, obwohl es erhebliche Anstrengungen und auch Fortschritte gegeben hat. Im Zentrum steht dabei der Übergang von der nichtformalen Welt des Anwenders mit seinen natürlich-sprachlichen Anforderungen in die formalisierte Welt des Informatikers mit seinen formalen (Programmier-)Sprachen. Dabei existieren ausgehend von den Requirements bis zum Software-Produkt und dessen Einsatz zahlreiche Herausforderungen und Schwierigkeiten, die eher praktischer denn theoretischer Natur sind, z.B. die mangelhafte Strukturierung von Informationen, das Vorhandensein redundanter, inkonsistenter Artefakte oder die fehlende Architekturdefinition.

In Verbindung mit der MDA stellt sich die Frage, ob die Modellierung und die Formalisierung die Lösung der Probleme des Software-Engineerings mit sich bringt. Die Möglichkeit dazu besteht sicherlich, allerdings sind einige wichtige Konzepte zu beachten. Besondere Aufmerksamkeit haben dabei die Artefakte, die sich in manuelle, generierte und historische Artefakte sowie in Ausgangs- und Zielartefakte unterteilen lassen. Durch Wiederverwendung und Automatisierung, den konsequenten Einsatz von standardisierten, formalen Sprachen sowie eine leistungsfähige Tool-Kette können manuelle und generierte Artefakte getrennt und die Konsistenz sichergestellt werden. Genau an dieser Stelle setzt die MDA der OMG an.

Anspruchsvolle Aufgabe für Software-Ingenieure: die MDA als Formalisierungs- und Automatisierungskonzept

Denn in erster Linie sollen die MDA und MDA-Tools dafür sorgen, dass durch Modelltransformationen Programmcode automatisiert generiert wird, anstatt ihn manuell zu erzeugen. Dabei ist Folgendes zu beachten:

- Der Formalisierung kommt eine Schlüsselrolle bei der Qualitäts- und Effizienzsteigerung zu. Bereits in frühen Phasen entstehen manuelle Artefakte mit hohem Formalisierungsgrad.
- Die Identifikation des Automatisierungspotenzials wird durch formale und semiformale Modelle ermöglicht bzw. erleichtert. Damit lassen sich generierte klar von manuellen Artefakten abgrenzen, was zur Beherrschung des Redundanzproblems führt.
- Entscheidungen bei Software-Projekten finden im Kontext des Gesamtsystems statt, d.h. kein »zufällig« entstehendes Design durch einzelne Programmierer. Manuelle Artefakte sind konsistent und verifizierbar: Sie sind durch Werkzeuge prüfbar und können weiterverarbeitet werden.

- Die Generalisierung (und Wiederverwendbarkeit) der Architektur durch modellgetriebene und architekturzentrierte Software-Entwicklung schafft Kapazität für das Wesentliche: die Erhebung und Formalisierung der Anforderungen.

Die MDA trennt die fachliche und die technologische Welt: Es entstehen zwei getrennte »Entwicklungswelten«.

Interessant ist sicherlich die Vision, endlich aus der chronischen Software-Krise herauszukommen und sich von den permanenten technischen Problemen derart zu lösen, dass die kurzen Innovationszyklen und die hohe Fluktuation der Techniken und Technologien sich zwar nicht verändern werden, sich aber die essenziellen Bereiche einer Software »herauslösen« lassen und somit eine Entkopplung stattfinden kann: Die formalisierte Essenz eines Systems kann unabhängig von einer technologischen Plattform in Form von Modellen entwickelt und verbessert werden, erst dann wird sie für die Generierung von Software verwendet. Ändern sich die Plattformen, kommt diese plattformunabhängige Essenz ohne Modifikationen aus, sondern sie ist lediglich mit einer anderen Plattform zu »kombinieren«, so dass dann die »neue« Software auf der Basis eines »alten« Modells entsteht.